ȘTEFAN TANASĂ, ȘTEFAN ANDREI, CRISTIAN OLARU

Java

de la 0 la expert

Ediția a II-a
revăzută și adăugită

POLIROM
2011

Seria *Web* este coordonată de dr. Sabin Buraga
(Facultatea de Informatică, Universitatea „Al.I. Cuza", Iași)

www.polirom.ro

Editura POLIROM
Iași, B-dul Carol I nr. 4; P.O. BOX 266, 700506
București, Splaiul Unirii nr. 6, bl. B3A, sc. 1, et. 1, sector 4,
040031, O.P. 53, C.P. 15-728

Descrierea CIP a Bibliotecii Naționale a României:

TANASĂ, ȘTEFAN

Java: de la 0 la expert / Ștefan Tanasă, Ștefan Andrei, Cristian Olaru. – Ed. a II-a rev. – Iași: Polirom, 2011

ISBN: 978-973-46-2405-8

I. Andrei, Ștefan
II. Olaru, Cristian

004.43 JAVA

Printed in ROMANIA

Cuprins

SQL permite şi definirea de tipuri de date utilizator, cunoscute şi sub numele de UDT (engl. *User Defined Types*) prin intermediul instrucţiunilor SQL `CREATE TYPE`. Acestea se împart în tipuri structurate şi tipuri distincte. Spre exemplu, prezentăm modul de creare a unui tip structurat de date:

```
CREATE TYPE PUNCT_PLAN
(
        X FLOAT,
        Y FLOAT
)
```

Un tip distinct poate fi creat având la bază un tip deja existent, aşa cum se poate vedea în exemplul următor:

```
CREATE TYPE NUMERE AS NUMERIC(10, 2)
```

Prin definiţie, un tip distinct SQL este asociat aceluiaşi tip ca şi tipul de bază. În exemplul prezentat, deoarece tipul `NUMERIC` este asociat cu `java.math.BigDecimal`, aceeaşi asociere va căpăta şi tipul `NUMERE`. Deci vom volosi `ResultSet.getBigDecimal()` pentru a obţine o valoare de acest tip.

8.6. Accesarea unei baze de date folosind JDBC

Procesul obţinerii de informaţii dintr-o bază de date folosind JDBC implică în principiu cinci paşi:

1. înregistrarea driverului JDBC folosind gestionarul de drivere `DriverManager`;
2. stabilirea unei conexiuni cu baza de date;
3. execuţia unei instrucţiuni SQL;
4. procesarea rezultatelor;
5. închiderea conexiunii cu baza de date.

8.6.1. Înregistrarea driverului JDBC folosind clasa `DriverManager`

Rolul managerului de drivere este acela de a ţine o referinţă la fiecare dintre obiectele driver disponibile în aplicaţia curentă. Un driver JDBC este înregistrat automat de managerul de drivere atunci când clasa driver este încărcată dinamic. Pentru încărcarea dinamică a unui driver JDBC, folosim metodele `Class.forName()`. După cum se observă, nu este nevoie de crearea unei instanţe a clasei driver odată ce aceasta a fost încărcată. Dacă se apelează metoda `newInstance()`, se va realiza un duplicat nefolositor al clasei driver. În exemplul următor vom încărca driverul punte JDBC-ODBC din pachetul `sun.jdbc.odbc` şi, mai apoi, driverul MySQL Connector/J, care permite conectarea la serverul de baze de date MySQL. Trebuie remarcat faptul că pentru Connector/J este necesar să includem în `CLASSPATH` calea spre pachetul care conţine driverul.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("com.mysql.jdbc.Driver");
```

`Class.forName()` este o metodă statică. Aceasta permite mașinii virtuale Java să aloce dinamic, să încarce și să facă o legătură la clasa specificată ca argument printr-un șir de caractere. În cazul în care clasa nu este găsită, se aruncă o excepție `ClassNotFoundException`. A se vedea `Java Reflection API`.

Driverele pot fi înregistrate și folosind metoda `DriverManager.registerDriver()`.

8.6.2. Stabilirea unei conexiuni cu baza de date

Odată ce s-a încărcat un driver, putem să-l folosim pentru stabilirea unei conexiuni cu baza de date. O conexiune JDBC este identificată printr-un URL JDBC specific. Sintaxa standard pentru URL-ul unei baze de date este:

```
jdbc:<subprotocol>:<nume>
```

Prima parte precizează că pentru stabilirea conexiunii se folosește JDBC. Partea de mijloc `<subprotocol>` este un nume de driver valid sau al altei soluții de conexiune a bazelor de date. Ultima parte, `<nume>`, este un nume logic sau alias care corespunde bazei de date fizice. Dacă baza de date va fi accesată prin internet, secțiunea `<nume>` va respecta următoarea convenție de nume:

```
//numegazda:port/subsubnume
```

În acest sens, un exemplu de adresă corectă este:

```
jdbc:mysql://localhost:386/arhiva
```

Prezentăm în continuare sintaxa particulară pentru câteva drivere JDBC:

ODBC – `jdbc:odbc:<sursa_date>[;<nume_atribut>=<valoare_atribut>]*`
MySQL – `jdbc:mysql://server[:port]/numeBazaDate`
Oracle – `jdbc:oracle:thin:@server:port:numeInstanta`

Pentru stabilirea unei conexiuni la o bază de date, se folosește metoda statică `getConnection()` din clasa `DriverManager`.

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/arhiva");
```

Pentru baze de date care necesită autentificare, se utilizează o formă a acestei metode cu trei argumente.

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/arhiva", nume_utilizator, parola);
```

Pentru a afla informații despre DBMS, va trebui să apelăm `Connection.getMetaData()` pentru a obține o instanță `DatabaseMetaData`, căreia îi putem aplica diverse metode de a afla informații despre tabelele bazei de date, sintaxa SQL suportată, dacă suportă sau nu proceduri stocate etc.

Folosind JNDI (*Java Name and Directory Interface*) putem să ne conectăm la o bază de date considerând-o sursă de date având un nume logic, în loc să codăm direct în aplicație numele ei și driverul pe care îl vom folosi. Acest mod de conectare nu este obiectul acestui capitol, fiind o caracteristică a sistemelor distribuite.

8.6.3. Execuția unei instrucțiuni SQL

După ce s-a stabilit conexiunea, se pot trimite instrucțiuni SQL către baza de date. API-ul JDBC nu verifică dacă instrucțiunea este corectă și nici apartenența ei la un anumit standard SQL, permițându-se astfel trimiterea chiar de instrucțiuni nonSQL. Programatorul este cel care știe dacă DBMS-ul interogat suportă interogările pe care le trimite și, dacă nu, el este cel care va trata excepțiile primite drept răspuns. Dacă instrucțiunea este SQL, atunci aceasta poate face anumite operații asupra bazei de date, cum ar fi căutare, inserare, actualizare sau ștergere.

API-ul JDBC specifică trei interfețe (pe care dezvoltatorul driverului trebuie să le implementeze) pentru trimiterea de interogări către bazele de date, fiecăreia corespunzându-i o metodă specială în clasa `Connection`, de creare a instanțelor corespunzătoare. Acestea sunt prezentate în tabelul care urmează:

Clasa	Metoda de creare	Explicații
`Statement`	`Connection.createStatement()`	Este folosită pentru trimiterea de instrucțiuni SQL simple, fără parametri.
`PreparedStatement`	`Connection.prepareStatement()`	Permite folosirea instrucțiunilor SQL precompilate și a parametrilor de intrare în interogări. Această metodă de a face interogări este utilă în cazul în care se fac interogări care diferă doar printr-un număr de parametri.
`CallableStatement`	`Connection.prepareCall()`	Permite folosire procedurilor stocate pe serverul DBMS.

Pentru execuția unei instrucțiuni SQL neparametrizate, se folosește metoda `createStatement()`, aplicată unui obiect `Connection`. Această metodă întoarce un obiect din clasa Statement.

```
Statement instructiune = con.createStatement();
```

Putem aplica apoi una dintre metodele `executeQuery()`, `executeUpdate()` sau `execute()` obiectului de tip `Statement` pentru a trimite DBMS-ului instrucțiunile SQL. Metoda `executeQuery()` este folosită în cazul interogărilor care returnează mulțimi-rezultat (instanțe ale clasei `ResultSet`), așa cum este cazul instrucțiunilor `SELECT`. Pentru operațiile de actualizare sau ștergere, cum ar fi `INSERT`, `UPDATE` sau `DELETE`, se folosește metoda `executeUpdate()` aplicată obiectului de tip `Statement`, rezultând un întreg care reprezintă numărul înregistrării afectate. Aceeași metodă este folosită pentru interogările SQL DDL, cum ar fi `CREATE TABLE`, `DROP TABLE` și `ALTER TABLE`, în acest caz returnând întotdeauna zero. Metoda `execute()` este

utilizată în cazul în care se obține mai mult de o mulțime-rezultat sau un număr de linie.

```
ResultSet rs = instructiune.executeQuery(
"select * from arhive");
String sql = "insert into arhive values (
'Popescu', 'Ion', 'Iasi')";
int raspuns = instructiune.executeUpdate(sql);
```

JDBC 2.0 permite trimiterea spre execuție a mai multor instrucțiuni SQL grupate (engl. *batch*), această facilitate mărind uneori performanțele. Prezentăm în continuare un astfel de caz:

```
Statement inter = con.createStatement();
con.setAutoCommit(false);

inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
int [] actualizari = inter.executeBatch();
```

Se observă dezactivarea modului *autocommit*. Acest aspect determină ca modificările rezultate în tabele după execuția metodei `executeBatch()` să nu fie automat salvate permanent (engl. *commit*) sau anulate (engl. *rollback*), aceste operațiuni rămânând la alegerea clientului. Se poate astfel ca în cazul în care una dintre interogări eșuează, clientul să anuleze efectele tuturor. Pentru alte informații, puteți consulta secțiunea dedicată tranzacțiilor. Metoda `executeBatch()` returnează un tablou în care elementele corespund interogărilor SQL efectuate și reprezintă numărul de linii afectate de instrucțiunile SQL corespunzătoare.

Pentru a executa independent instrucțiunile SQL, se poate păstra modul *autocommit* folosind captarea excepțiilor `BatchUpdateException` aruncate în caz de eroare, așa cum se poate vedea în exemplul următor:

```
try {
   inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
   inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
   inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
   int [] actualizari = stmt.executeBatch();
} catch(BatchUpdateException b) {
      System.err.println("Actualizari realizate: ");
      int [] eroriActualizari = b.getUpdateCounts();
      for (int  i = 0; i < eroriActualizari.length; i++) {
            System.err.print(eroriActualizari[i] + "  ");
      }
      System.err.println("");
}
```

Ștergerea comenzilor dintr-un grup se realizează cu metoda `clearBatch()`.

Un driver JDBC poate să nu ofere suport pentru execuția grupată a unor instrucțiuni SQL. Pentru a afla dacă e permis acest lucru, se va folosi metoda `supportsBatchUpdates()` din clasa `DatabaseMetaData`.

Dacă se dorește realizarea de apeluri SQL având date variabile drept intrare, se va folosi clasa `PreparedStatement` care moștenește clasa `Statement`. Prezentăm în continuare modul de construcție a unei astfel de instrucțiuni, unde `con` reprezintă o instanță `Connection`:

```
PreparedStatement  instructiune = con.prepareStatement(
"update arhive set nume = ? where prenume like ?");
```

După cum se observă, prin construcție, obiectul de tip `PreparedStatement` primește ca intrare o instrucțiune SQL (spre deosebire de cazul `Statement`). În momentul construcției, instrucțiunea SQL primită ca parametru este trimisă direct spre DBMS, unde este precompilată. Mai rămâne să dăm valori variabilelor folosind metode `setXX()` corespunzătoare tipurilor de date și să executăm efectiv interogarea, după cum se poate vedea în continuare:

```
instructiune.setString(1,"Popescu");
instructiune.setString(2,"Ion");
instructiune.executeUpdate();
```

Această manieră de interogare este mai rapidă decât folosirea clasei `Statement`, în cazul în care dorim execuția repetată a unei instrucțiuni SQL. Instrucțiunea SQL este compilată o singură dată în momentul creării instanței clasei `PreparedStatement` și folosită apoi repetat, eventual cu valori diferite ale parametrilor. Se poate folosi și în cazul în care nu avem parametri, după cum se observă în continuare:

```
PreparedStatement  instructiune = con.prepareStatement(
"select * from arhive");
ResultSet rs = instructiune.executeQuery();
```

Nu vom prezenta în acest capitol procedurile stocate datorită nivelului relativ ridicat de complexitate.

Metodele fără parametri de intrare puse la dispoziție de clasa `Connection` pentru cele trei interfețe prezentate în tabelul precedent produc mulțimi-rezultat fără cursor deplasabil și nesenzitive la modificări. O mulțime-rezultat are cursor pentru poziționare deplasabil, proprietate care poartă numele de *scrolling*, dacă deține un pointer interior care indică înregistrarea accesată la momentul curent, pointer ce se poate deplasa programatic. De asemenea, după cum spuneam, mulțimea-rezultat nu este senzitivă (engl. *sensitive*) la modificările care pot surveni între timp în tabela interogată. Prin senzitivitate se înțelege actualizarea automată a mulțimii-rezultat pentru a ilustra dinamic modificările survenite între timp în tabelă.

Driverele care sunt conforme cu versiunea 2.0 a specificației JDBC permit obținerea de mulțimi-rezultat senzitive și având cursor deplasabil. Acest lucru este posibil prin specificarea în constructorul instanței `Statement` a uneia dintre constantele prezentate în tabelul care urmează:

Constantă	Explicaţii
TYPE_FORWARD_ONLY	Acest tip de ResultSet este cel din JDBC 1.0. Mulţimea-rezultat nu are cursor deplasabil decât dinspre început spre sfârşit. Modificările făcute în tabelă nu se reflectă în mulţimea-rezultat.
TYPE_SCROLL_INSENSITIVE	Mulţime-rezultat *scrollable*, deci cursorul poate fi mutat înainte şi înapoi sau pe orice poziţie diferită de poziţia curentă. De asemenea, eventualele modificări făcute între timp în tabelă nu sunt reflectate în ResultSet.
TYPE_SCROLL_SENSITIVE	Mulţime-rezultat *scrollable*. Senzitivitate la modificări.

De asemenea, driverele conforme cu specificaţia 2.0 a JDBC oferă şi posibilitatea actualizării programatice a tabelelor prin intermediul obiectelor ResultSet. Tipul de actualizare se specifică prin intermediul uneia dintre constantele prezentate în continuare:

Constantă	Explicaţii
CONCUR_READ_ONLY	ResultSet-ul nu poate fi modificat programatic. Oferă posibilitatea unui număr nelimitat de conectări concurente la baza de date, deoarece nu se fac modificări asupra tabelei care ar putea genera conflicte. Acest tip de ResultSet este specific driverelor conforme specificaţiei JDBC 1.0.
CONCUR_UPDATABLE	ResultSet-ul şi baza de date pot fi modificate programatic. Sunt posibile un număr limitat de conexiuni concurente dat de tipul de concurenţă ales (a se vedea secţiunea dedicată tranzacţiilor). Acest tip de ResultSet este specific driverelor conforme specificaţiei JDBC 2.0.

Prezentăm un exemplu de cod care va determina crearea unei mulţimi-rezultat având cursor deplasabil şi senzitivă la modificări (ordinea parametrilor metodei createStatement() este importantă).

```
Statement declar = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.
    CONCUR_UPDATABLE);
// la fel se poate crea o instanta a clasei PreparedStatement
ResultSet rez = declar.executeQuery(
    "select nume, prenume from arhive");
```

8.6.4. Procesarea rezultatelor

Pentru parcurgerea simplă a înregistrărilor unui obiect din clasa ResultSet folosind JDBC 1.0, putem folosi metoda next(), ca în următoarea secvenţă de cod.

```
while (rs.next()) // implicit, cursor pozitionat inainte de prima
```